CS 4444 Spring 2018 – Due in class February 6

Assignment 1: Sequential Program Optimization

Introduction

This assignment focuses on optimizing a sequential program – you gain very little by running an inefficient program on a parallel computer.

Files needed

To start, download from the course Collab site the following files.

- hw1.pdf: this file
- generate_input.c: this program will be used to generate the input (hence its name). You don't need to know C to use this file.
- original.c or original.f: this is the file that you will need to optimize. Pick your flavor depending on whether you want to code in C or Fortran.

The program

The program we will be working on takes in an input file that specifies a series of (atomic) particles – their (x,y,z) coordinates and a charge. The program then computes the following fictional function on the data:

$$E = \sum_{j<i} \left(\frac{e^{(r_{i,j} * q_i)} * e^{(r_{i,j} * q_j)}}{r_{i,j}} - \frac{1}{a} \right) \qquad \text{if } r_{i.j} \leq cut$$

Where q_i and q_j are the charges of the particles, a is a constant (defined as 3.2), $r_{i,j}$ is the distance between the two points, and is defined as $r_{i.j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$, and *cut* is the cut-off, defined as the second command-line parameter (we'll be using 0.5 for that value for this entire homework, although you cannot assume that in your code).

Generating the input

The program (in original.c or original.f) takes in a series of coordinates (and the charge for the particle at those coordinates) from an input file, and computes the value of the formula above. The first line of the input file is a single integer, which is the number of particle coordinates specified in the file. Each additional line contains one atomic particle specified by four numbers: the (*x,y,z*) coordinates and the charge. Each of these 4 values is a single number between 0 and 1, and are separated by spaces (we're assuming for simplicity sake here that you can have particles with a non-integer amount of charge).

First we need to generate the input file. You'll need to compile the generate_input.c program as such:

```
gcc –o generate_input generate_input.c
```

A note for those who know C: using an optimization flag (such as –O3) won't make any difference here – the time taken is dependent on the file I/O and the rand() routine, which (being in the system library) is already optimized.

To generate an input file, you will need to specify one or two parameters. The first parameter is how many coordinates to generate. The second parameter is an optional random seed.

To generate a file containing 100 coordinates, enter the command:

```
./generate_input 100 > input.txt
```

This will create an input.txt file that contains 100 coordinates in the input file format needed for original.c/original.f. Each time you run the generate_input program, it will generate a different set of 100 coordinates. If you want to generate the same set of coordinates each time (useful for comparing your program to somebody else's), you should enter a random seed, which can be any integer value. For example:

```
./generate_input 100 5 > input.txt
```

This will generate the exact same set of coordinates each time the program runs. Note that running the generate_input program with the same seed but on different computers will not necessarily create the same input file.

Note that an input size of 100 is good to start working with. As you want to test your program for efficiency, you will want a larger input size. The algorithm is an n^2 algorithm – meaning that each time you double the input, the running time is quadrupled.

The task

The task for this assignment is to make the original.c or original.f program run as efficiently as possible. When you run the program, you will see the following output (this is the output from the C version of the program; the Fortran output is formatted slightly differently, but has the same information). The input file was created with a size of 10,000 and with a random seed of 10; the cut-off was set at 0.5.

```
Value of system clock at start = 0
Coordinates will be read from file: input.txt
Natom = 10000
cut =     0.5000
Value of system clock after coord read = 40000
Value of system clock after coord read and E calc = 5610000
                        Final Results
                        -------------
                  Num Pairs = 13783190
                    Total E = 57845101.4740826413
     Time to read coord file =         0.0400 Seconds
         Time to calculate E =         5.5700 Seconds
        Total Execution Time =         5.6100 Seconds
```

First, make sure that your computed value of *E* is the same – otherwise, your program is not properly computing the value anymore.

Be wary that different C compilers (and possibly Fortran as well) *will* compute a different value for *E*: gcc, icc, and pgcc (all with –O3) helpfully compute completely *different* results for *E*

for the exact same original.c program: the values computed are 57845094.7397084311, 57845101.4740826413, and 57845064.8572922871, respectively. gcc is the real winner here, as it will compute a different value for *E* for different optimization levels (no optimization computes 57845101.4740827754).

The times you want to look at to compare trial runs is both the time to calculate *E* and the total execution time. The first is how much time the algorithm you are optimizing took, whereas the second includes the initialization and reading in the input file.

Any optimization is valid, as long as your program computes the same result on various inputs as the original program. Note, however, the lines of code that start and stop the timing of the program must still encompass all your work – in particular, the setting of the time0 variable must come before any computations, and the setting of the time2 variable must come at the end (you can deallocate the arrays after the setting of time2).

You should also use the optimization flag (–O3) when compiling your program:

```
gcc –lm –O3 –o original original.c
```

In general, the –O3 flag is the minimum you should use. Nearly all compilers have a wide variety of optimization flags available. Examine the manual page carefully for your compiler, e.g.

```
man gcc
```

Look for the section on optimization. Experiment with different flags. But make sure the same result is computed! Although the –O[N] flag is nearly universal across Unix compilers, other optimization options are compiler-dependent, and even the level of optimization corresponding to N can vary from compiler to compiler.

Prepare a table in your write-up with timings for the original code with no optimization, -O1, -O2, and –O3, as well as for each of your by-hand optimizations.

Running the program – Remember use the PBS SLURM to execute your job

To run the program, enter the following:

```
./original input.txt 0.5
```

That specifies that the input file is called input.txt, and the cut-off is 0.5. If you create the input file as described above (size 10,000 and random seed 10), then you should get the same results as shown above.

For the final testing – to see how efficient everybody's algorithm is – we will use an input size of 20,000 with a random seed of 1 and a cut-off of 0.5. The unoptimized versions take about 22 seconds to complete this execution run (resulting in 55,381,678 pairs with a value of *E* equal to 232,735,477.6827850044).